Towards Unified Authorization for Android

Michael J. May, Karthikeyan Bhargavan

Towards Unified Authorization for Android

Michael J. May[1]* and Karthik Bhargavan[2]

[1] Kinneret College on the Sea of Galilee, DN Emek Hayarden 15132, ISRAEL
[2] INRIA

Abstract. Android applications that manage sensitive data such as email and files downloaded from cloud storage services need to protect their data from malware installed on the phone. While prior security analyses have focused on protecting system data such as GPS locations from malware, not much attention has been given to the protection of application data. We show that many popular commercial applications incorrectly use Android authorization mechanisms leading to attacks that steal sensitive data. We argue that formal verification of application behaviors can reveal such errors and we present a formal model in ProVerif that accounts for a variety of Android authorization mechanisms and system services. We write models for four popular applications and analyze them with ProVerif to point out attacks. As a countermeasure, we propose Authzoid, a sample standalone application that lets applications define authorization policies and enforces them on their behalf.

1 Introduction

The Android operating system seeks to foster a rich ecosystem of third-party applications. Users may download apps from reputable stores managed by Google and Amazon or directly from app developers.[3] This leaves users vulnerable to malware masquerading as genuine apps. Consequently, Android provides strong runtime isolation, running each application process in a separate Dalvik virtual machine, and giving each a private storage area. Isolated applications may still share files and data, for example using external storage or using an inter-app messaging mechanism called *intents*. While some apps freely share and collaborate with others, those holding sensitive data are tempered by the need for security and integrity. Android therefore provides authorization mechanisms which let an app control which other apps, if any, can read or write its data.

System Permissions Android protects its system resources through *permissions* which are granted by the user at installation time and accompany the app throughout its lifetime. The Android SDK defines about 130 built-in permissions of which some forty are *signature/system* permissions and are reserved for the operating system or apps installed by the device manufacturer [28]. The rest can be requested by an app in its *application manifest*, an XML file which

* Work performed while visiting INRIA.
[3] It is estimated that the average Android phone in 2012 has 32 apps installed [24].

is prepared by the developer and stored in its application package (APK) file. When an app attempts to access a system API function at run time, Android first checks if the requestor has the required permission. If it doesn't, a security exception is thrown.

Some examples of regular permissions are: `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions to read or write to the phone's shared disk (referred to informally as the *SD card* since its default mount point is `/mnt/sdcard/` [21]), `INTERNET` to open network sockets, and `READ_LOGS`[4] to access the system log. Some examples of system permissions are: `INSTALL_PACKAGES` to install new applications, `BRICK` to disable the phone completely, and `DELETE_CACHE_FILES` to clear the cache directories of other apps.

Application-level Authorization In addition to install-time permissions, Android provides a variety of other authorization mechanisms. Activities can filter which intents will be directed to them based on the intent's content or requested action. Content providers and services can be made available only to applications which have certain permissions. Authorization may seem seamless to the user, but due to the variety of tools available and the details of the OS, it can be technically messy and sometimes can even be bypassed.

Consider, for example, a user who installs Dropbox (`www.dropbox.com`), uses it to download a PDF file from the cloud, and opens it with Adobe Reader (`adobe.com/products/reader.html`).[5] The user would assume that during the transaction Adobe Reader got temporary read access to the file and nothing more. As we discuss later, that is not the case at all: Adobe Reader and (up until API level 16) *all* of the applications on the phone can read the file indefinitely afterwards. Many can modify it too.

Our Contribution There are many ways in which applications get authorization wrong or fail to enforce authorization properly. They fail primarily because they don't define the policy they are trying to enforce and (likely) didn't use a full model of the environment during testing. Proper modeling of the environment and the application's behavior would reveal attacks on the authorization mechanism.

Our contribution in this work is threefold. First, we present a unified picture of the Android authorization tools, something not previously presented in a single work. Second, we show how many popular sharing applications on the market fail to get authorization right and publish a formal model of the Android authorization tools and environment which allows us to reveal attacks. We publish the model so that others can use and extend it to test their apps. Third, we present Authzoid, a sample authorization app which properly implements the authorization tools that the apps got wrong. The code can act as a source code module to be included as is or as a starting point for developers who want to get authorization right. Both code and model are found at: `http://prosecco.gforge.inria.fr/Essos/pv/`.

[4] Changed to signature/system/development permission in API level 16.
[5] The most popular PDF reader on Google Play as of Oct 2012.

The rest of this paper is organized as follows. Section 2 explains the Android authorization tools. Section 3 discusses our attacker model, gives technical descriptions of the sharing applications surveyed, and explains the attacks against them. Section 4 contains our formal model for Android authorization tools, environment, and the applications studied. Section 5 discusses the Authzoid app and its major features. Section 6 contains related work and Section 7 concludes.

2 Authorization for Android Applications

Android applications are composed of four kinds of run time entities:

Activities correspond to windows and allow for user interaction via a GUI.
Content Providers provide SQL-like interfaces to queryable data.
Broadcast Receivers listen for broadcast messages from other application components, the operating system, or other applications.
Services run in the background and provide long term functionality without providing a user interface.

Applications exchange messages via intents which contain a URI data field and strings, URIs, or key-value pairs in *extra* fields. They are routed by an Android component called *Binder* between the run time entities. During routing, the *Activity Manager* writes the action, sender, recipient, and data field (but not the extra fields) to the log.

Each runtime entity may use a variety of authorization mechanisms to control access to its data and functionality. In the rest of this section we review the five authorization mechanisms available and explain their usage, strengths, and weaknesses. For each mechanism, we give an example of how it is used in a popular application currently available from the Android Market. Some apps use a combination of the mechanisms below to enable a variety of user policies.

Android Permissions Android's SDK includes about 130 permissions, but an app may extend them with its own permissions by adding them to its manifest file. Apps can use permissions to enforce authorization in one of two ways.

First, content providers, activities, services, and broadcast receivers can specify that only applications with a certain permission may access them. For activities and services, this prevents applications without the permission from invoking or binding to them. For content providers, separate read and write permissions may be given. For broadcast receivers, it prevents the delivery of broadcasts from apps without the permission. The filtering is done automatically by Binder based on the app's manifest file (`android:permission` for activities, services, and broadcast receivers and `android:readPermission` and `android:writePermission` for content providers) or as defined programmatically if they are configured in code.

Example: K-9 Email (`code.google.com/p/k9mail/`) uses the custom permissions `com.fsck.k9.permission.READ_ATTACHMENT` to protect its attachments content provider. Its email messages content provider protects read access with `READ_MESSAGES` and write access with `DELETE_MESSAGES`. GMail (`gmail.com`) and Yahoo! Mail (`mail.yahoo.com`) (discussed below) use a similar strategy.

Second, apps can use the system API to discover whether it, the app which called it, or another app has a particular permission (using `checkPermission()` or `checkCallingPermission()`), regardless of its type. They can then make programmatic decisions based on the results.

Example: Some plug-in libraries (*ex.* ACRA (`code.google.com/p/acra/`)) programmatically investigate which permissions are available to their host applications before attempting actions which require particular permissions (*ex.* reading the log and sending internet data).

URI Permissions The content provider read and write manifest permissions give blanket read and write access. Alternatively, a content provider can give specific read or write query access to a single `content` URI under its authority. The URI permission can be granted programmatically using `grantUriPermission()` or by sending an intent to the recipient with the `FLAG_GRANT_READ_URI_PERMISSION` or `FLAG_GRANT_WRITE_URI_PERMISSION` flags set. URI permissions can be delegated by recipients. Intent-granted URI permissions are valid until the recipient app closes or is killed. Programmatically-granted ones are valid until revoked using `revokeUriPermission()`.

Binder enforces URI permissions by tracking the grants, revokes, and intents sent, so the URI does not need to be secret. Also, since intents can be routed by capability, the sender may not know which app received the permission.

Depending on whether the `content` URI refers to a database row, a file, or both, the recipient can use a *content resolver* request to read or write the corresponding rows or file. If the URI is opened as a file using `openFile()`, the content provider returns an open file descriptor for it and Binder assigns ownership of it to the recipient.

Example: Users can open an attachment from K-9 Email with an external viewer. When this happens, K-9 Email sends an intent to the viewer with a `content` URI for the attachment and the URI read permission flag set. The recipient can then use a content resolver to resolve the URI to an open file descriptor. GMail and Yahoo! Mail employ a similar strategy.

The use of open file descriptors leads to some technical inconveniences. First, since a file descriptor is a hard link to the file and is owned by the recipient, the sender can't close the file descriptor or delete the file until the recipient closes it. Second, if two application hold open file descriptors for the same file (*i.e.* they both requested the same URI), they cause read/read and read/write conflicts and race conditions. Third, only a few classes in the Java file API support file descriptors, making it impossible to perform random access reads

or writes to the file and making rewinding difficult. Because of these issues, some applications immediately make local copies of files passed to them by URI (*ex.* Adobe Reader) or don't enable updates to such files (*ex.* Jota Text Editor (`sites.google.com/site/aquamarinepandora/home/jota-text-editor`)).

Private Storage Every Android application is given its own user name, group name, and home directory. The home directories are protected by Linux file and directory permissions and by default no app can read or write the home directory of another. Apps can override the default settings to make files or directories world readable, writable, or executable using `setReadable()`, `setWritable()`, and `setExecutable()`. Then any other app can read, write, or execute the files or directories. If an app makes a file world readable in order to share it, it may include a long random string in the path to make it hard for unintended apps to guess the path name. This technique turns the path name into a secret, so the app must ensure that only the intended recipient gets the path name.

Example: Unlike most apps which keep all files in private storage private, Google Drive (`drive.google.com`) (discussed below) selectively sets path read and execute permissions to enable others to read files in its private storage.

External Storage (SD Card) Most Android devices have a shared storage space for files or data (the SD card). Read and write access to the SD card require the permissions mentioned above. Many applications (*ex.* the camera, the default browser's downloads folder) use the SD card for storing files that are either too large to keep in private storage or that are meant to be available for other apps to use. Aside from the read and write permissions, Android does not enforce access control on the SD card, so any application can read, write, or delete any file on it. Authorization can be enforced on the SD card using encryption or message authentication codes (MAC).

Example: The password storage app 1Password (`agilebits.com/onepassword`) stores its encrypted password database on external storage. It doesn't share passwords directly with other apps, instead using the clipboard to copy and paste passwords. Encryption protects the contents of the password database and MACs protect its integrity.

Web Sharing Some apps place data to be shared on a public web site and send the URL for the data to another app via an intent. Often the URL contains a long random string to make it difficult to guess, turning the URL into a secret. Another option is to protect the URL using web-based application or user authentication such as OAuth [14].

Example: MyTracks (`www.google.com/mobile/mytracks/`) uses GPS information to track where the device has gone, including distance traveled, speed, and elevation change. When sharing a "track" from MyTracks with another app, it first uploads a custom map to Google Maps and then sends a web URL with a long random part to the recipient via an intent.

Summary The authorization tools listed show the variety of mechanisms available. It's not clear if any or all of the tools are sufficient to achieve a satisfactory authorization policy. The applications that we study in the next section use different combinations of the tools to enforce authorization, but each suffer from attacks and weaknesses that demonstrate that using them correctly is not simple. In some, the tools are used incorrectly; in others, features of the Android environment defeat the app's authorization policy.

3 Applications and Attacks

To investigate how popular applications use the authorization tools of Section 2 to enforce their security goals, we study four apps: two Email clients and two Cloud File Storage applications. We explain each application's authorization mechanism and explain how an attacker may defeat it.

3.1 Authorization Goals and Attacker Model

Since the apps we examine don't specify authorization policies in their documentation, we define a minimal one for the purposes of our study. Our minimal policy contains just one confidentiality rule and one integrity rule:

Confidentiality An app may read a file only if it owns it or if the owner and the user have authorized the reading.

Integrity An app may modify a file only if it owns it or if the owner and the user have authorized the modification.

The policy can be enforced by many authorization mechanisms, including those listed in Section 2.

Regarding the attacker model, we first assume that the Android protection mechanisms are enforced according to their specification (*i.e.* the phone isn't "rooted", giving arbitrary power to an app). Next, we make the same assumption that Android does regarding app isolation: that apps are mutually suspicious. The attacker is assumed to be (1) installed on the phone, (2) capable of performing polynomial time programmatic tasks, and (3) in possession of a set of authorization-related permissions that seem may seem innocuous: `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE` (51% of popular applications request it), `READ_LOGS` (6% of popular applications request it[6]), and `INTERNET` (77% of popular applications request it) [10]. Any series of actions which such an attacker can take to contravene the authorization policy defined above is an attack.

[6] As of Oct 2011, `READ_LOGS` was the ninth most popular dangerous Android permission requested. In API level 16, it was converted to a system/signature/development permission, so access to it on the most recent devices is significantly reduced.

3.2 Study of Sharing Applications

We now consider four popular Android applications which enforce authorization using the mechanisms defined in Section 2. The applications are chosen because they illustrate the use of a variety of mechanisms and are representative of classes of apps.

GMail downloads attachments to its private storage area and manages them via a content provider which is protected by custom permissions `READ_GMAIL` and `WRITE_GMAIL`. The permissions are signature level permissions, so only Google applications can request them [7]. GMail allows the user to open an attachment using an outside document viewer by sending an intent containing a URI read permission. Applications which behave similar to GMail include the built in Android Email application and K-9 Mail.

Attacks: The use of a protected content provider ensures that only applications sent the URI permission can read the file. However, some recipient viewers immediately make a copy of any file sent to them by URI. For example, Adobe Reader copies any file it shows to the SD card in the downloads directory, making it readable by an attacker (confidentiality).

Yahoo! Mail has a content provider which is protected by a custom signature permission (`com.yahoo.mobile.client.android.permissions.YAHOO_INTER_APP`). Yahoo! lets the user open attachments using URI permissions, just like GMail. However, downloaded attachments are stored on the SD card, so they are readable by any application with `READ_EXTERNAL_STORAGE`. The MailDroid (`groups.google.com/group/maildroid`) application behaves similarly.

Attacks: Since Yahoo! stores all downloaded attachments on the SD card, an attacker can read them (confidentiality). The application does not check for downloaded file integrity, so once on the SD card they may be modified by an attacker as well (integrity).

Google Drive offers two mechanisms for sharing files on the phone.

First, the on-phone app lists the files and directories on the cloud and downloads one when the user requests to view or share it. Files can't be updated on the phone. A downloaded file is placed in a new, randomly named directory in a document cache directory located in Google Drive's private storage (`/data/data/com.google.android.apps.docs/cache/`). The new directory contains just one file and is world readable and executable. The file is made world readable and all the directories above the randomly named directory are world executable, letting any app open the file, but not list the directory names under `cache`. The file path is sent to the target as the `data` field of an intent.

Second, apps can access Google Drive via a web service interface which is protected by SSL and OAuth 2.0. An app receives an API identifier which it

can use to obtain file read and write tokens. An app can download files via their names or identifiers and send updates back over the web.

Attacks: With respect to the on-phone app, the directory path is a secret since any application which knows it can open the file. Activity Manager, however, prints the `data` fields of all intents to the log, so the path is printed as well. An attacker which has `READ_LOGS` permission can discover the path and read the file (confidentiality).

Dropbox lists the files and directories in the cloud and downloads them on demand. The files are stored on the SD card in a directory called `scratch`. When sharing a file, it is first stored in the `scratch` directory and then the path and filename are sent via an intent.

Downloaded files can be opened for reading and editing, but are not checked for integrity after downloading. When opening a file, the user chooses which file viewer to use; if the viewer saves a new version, it is uploaded to the cloud. Saves are monitored until the authorized viewer closes or loses focus. Dropbox ignores saves by other applications, even when an authorized viewer is working.

When sharing a file for attachment to an email, the file is uploaded (if necessary) and a web URL is provided in the intent as an extra. The URL provides read only access to the file and includes a random string to make guessing harder.

Attacks: Since downloaded files are stored on the SD card, they are readable by an attacker (confidentiality). Unauthorized saves are not automatically uploaded to the cloud, but since there is no integrity check, an attacker can tamper with a file and subsequent views of the file on the phone will show the tampered version. If a viewer unknowingly saves a tampered version, the attacker's modifications will reach the cloud (integrity).

3.3 Discussion

Our study of four popular and well-regarded applications illustrates the difficulty in getting even a simple authorization policy right. Many applications place sensitive data on public external storage. Some use unguessable directory names in private storage, but these names may leak into the shared system log. Still others may themselves correctly implement access control, but may be let down by the applications with which they share files.

Simple technical tricks aren't sufficient against a dedicated adversary. Wuala (`wuala.com`) tries to place shared files on the SD card for only a few seconds. However, due to Android's application life cycle, a malicious app can monitor the SD card and breach confidentiality during that gap. Boxcryptor (`boxcryptor.com`) encrypts files on the SD card and decrypts them just in time for the recipient. Such uses of encryption are limited by key management, that is, how to securely transfer a secret key to the recipient. Google Drive's example shows that transferring secret keys by intent is not always secure. Keys derived

from passphrases are hard to keep secret, as shown by Belenko and Sklyarov [3]. Even if encryption keys are shared securely, file storage applications often misuse encryption and integrity algorithms and expose their plaintext to attackers [4].

We advocate a unified comprehensive approach to the implementation of application-level authorization. Rather than suggest point-fixes to prevent specific attacks, we show how to write formal models that precisely capture authorization policies and relevant parts of the execution environment. By automatically analyzing such models, we can both find attacks and gain confidence in the mechanisms used to enforce the policy.

4 Formal Model

As shown above, implementing even a minimal authorization policy requires an analysis of the authorization tools as well as the environment. Modeling can help such an analysis by including relevant parts of the Android authorization tools and the operating system. Developers can then create a model of their application, run it inside the Android model, and use automated tools to discover attacks. In this section we describe the building of such a model using ProVerif [5]. We show illustrative snippets of its parts: (a) the authorization policy, (b) the Android authorization tools, (c) parts of the Android OS, and (d) the sharing application. We then use the model to discover attacks in the models of the applications surveyed. ProVerif is well suited for our needs since (1) it enables the definition of authorization policies using Horn clauses and communication using the applied pi calculus; (2) it can model enforcement mechanisms that use secret and fresh file or path names and cryptography; and (3) it lets us analyze the models against an unbounded adversary.

Policy Language The snippet below implements the minimal authorization policy from Section 3.1. It allows an app to read or write files only if it is the file's owner or if it receives authorization from the owner and the user. Lines 1–2 are a horn clause saying that if an application (a1) and a user (u) own a resource (r), then a1 is authorized to read r. Lines 3–4 enable another application (a2) to receive read authorization from the owners (a1 and u). The parallel write rules are omitted due to space considerations.

```
1 clauses forall u:Principal, a1:appid, a2:appid, r:resource;
2       owners(r, u, a1) -> readAuthorized(a1, r).
3 clauses forall u:Principal, a1:appid, a2:appid, r:resource;
4       owners(r, u, a1) && userAuthorizedRead(u, a2, r) -> readAuthorized(a2, r).
```

Android Authorization Tools We implement the following Android authorization tools:

Permissions are included via a `androidPerm` type which is populated with permissions that can be granted by the user during installation.

URI Permissions are included via a `uri` type which refers to a file resource. Resolution is modeled using a lookup table of `uri` and `resource` pairs.

Private Storage is modeled by using a `path` type which refers to a location only accessible by the owner. If the path is declared world readable, writable, or executable, others can access it too. Fresh path names may be world readable, but only can be accessed if the requestor knows the path's name.

SD Card is modeled as a file system process which enables the storage or retrieval of objects based on `path` and `filename` objects stored in a lookup table.

Web Sharing is parallel to private storage, but without the need for setting path permissions.

The following snippet shows how file and log read permissions are handled (parallel file write clauses are elided). Lines 5–7 define the Android permission type, the permission to read the SD card (externalRead), and the permission to read the log (logRead). Lines 8–9 allow an application a to read a file with name f, path p, and any file and path permissions fp and pp if (1) a has the external read permission and (2) the file is on the SD card. Lines 10–11 allow an application a to read all files in its own private space (private(a)). Lines 12–14 allow an application a to read a file in another application o's private space if its path permissions (pp) are set to world executable (isWorldExecutable) and its file permissions (fp) are set to world readable (isWorldReadable). Line 15 allows an application a to read the log if it has logRead.

```
5  type androidPerm.
6  fun externalRead() : androidPerm.
7  fun logRead() : androidPerm.
8  clauses forall a:appid, l:location, p:path, f:filename, pp:filePerms, fp:filePerms;
9   hasPermission(a, externalRead()) -> canReadFile(a, sdcard(), p, pp, f, fp).
10 clauses forall a:appid, l:location, p:path, f:filename, pp:filePerms, fp:filePerms;
11  canReadFile(a, private(a), p, pp, f, fp).
12 clauses forall o:appid, a:appid, p:path, f:filename, pp:filePerms, fp:filePerms;
13  isWorldExecutable(pp) && isWorldReadable(fp) ->
14  canReadFile(a, private(o), p, pp, f, fp).
15 clauses forall a:appid; hasPermission(a, logRead()) -> canReadLog(a).
```

Android OS Elements We include processes for the following authorization related Android processes:

File System process which enables applications to read, write, and list files on the SD card based on path and file name. The file system allows access to files in private storage by the owner and by others which know the path name if the permissions are set correctly.

Content Provider process which enables applications to resolve URIs and thereby read or write files which they refer to.

Binder process which handles the granting of URI permissions, both from the owner and via delegation. Binder writes entries in the log.

Log process which gives permission-based read and write access to the log.

Permission Granting process which enables the user to grant permissions to processes.

The following snippet shows three parts of file system's code in the model. Lines 16–17 listen for file read requests (readFile) and check the application is registered. Lines 18–19 retrieve the file based on its location (l), path (p), and file name (f), check if a is able to read it, and return it to a if it is able. Lines 20–22 listen for requests to list the files in a directory path. Line 23 allows it if the path is world readable. Lines 24–27 allow an application to list all files on the SD card if the requestor has externalRead permission.

```
16 let FileSystem() = (!in (filesystem,readFile(a, l, p, f));
17    get apps(=a) in
18    get files(=l, =p, pp, =f, fp, r) in
19    if canReadFile(a, l, p, pp, f, fp) then out(return(a), r))
20  | (!in (filesystem,listFile(a, l, p));
21    get apps(=a) in
22    get files(=l, =p, pp, f, fp, r) in
23    if isWorldReadable(pp) then out (return(a), f))
24  | (!in (filesystem,listSDCard(a));
25    get apps(=a) in
26    get files(=sdcard(), p, pp, f, fp, r) in
27    if hasPermission(a, externalRead()) then out (return(a), (p, f))).
```

Testing Application We implement a single process for each sharing application. The process registers the application, specifies how files are added, and specifies how files are shared with other applications ("open with").

The following snippet shows the Dropbox application. Line 28 defines the Dropbox application id as private (not known to the attacker initially). Line 29 is the header for the process. Lines 30–31 register Dropbox in the applications table (apps, definition elided) and publish the name of its private storage (private(dropbox)) and its web storage (web(dropbox)) by sending their values on the free channel pub (definitions of private, web, and pub are elided). This simulates an attacker knowing the application's root directory and web domain, but not knowing the paths below them where files are found. Lines 32–34 define a new file's contents (r), its file name f, and its path p; assigns ownership of the file to Dropbox (line 33, using the assume function which is elided); and inserts it in the files table (definition elided) on the web (web(dropbox)) where it stays until downloaded. The noPerms() terms are used to model file and path read, write, and execute permissions. Lines 35–40 model a user opening a file. Lines 35–36 receive an openWith command and download a file from Dropbox's web space (path p, path permissions pp, file name f, file permissions fp, file contents r). Line 37–38 select an application a and authorize it to read. Line 39 stores

the file on the SD card in the files table with no explicit file or path permissions (noPerms). Line 40 returns the path and file name to the requestor by an explicit intent (explicitintent(a)).

```
28 free dropbox : appid [private].
29 let Dropbox(u:Principal) =
30    (insert apps(dropbox);
31    out (pub, (private(dropbox), web(dropbox))))
32  | (!new r:resource; new f:filename; new p:path;
33    if assume(owners(r, u, dropbox)) then
34    insert files(web(dropbox), p, noPerms(), f, noPerms(), r))
35  | (!in (openWith, ());
36    get files(=web(dropbox), p, pp, f, fp, r) in
37    get apps(a) in
38    if assume(userAuthorizedRead(user, a, r)) then
39    insert files(sdcard(), p, noPerms(), f, noPerms(), r); (*readable by attacker*)
40    out (explicitintent(a), (p, f))).
```

Discovery of Attacks By combining the testing application's model with the environment and authorization code, we can check two types of queries in ProVerif:

1. Checks that proper authorization is reachable. ProVerif should show traces by which a user can properly authorize an app to read and write a file.
2. Checks that an attacker can't read or write files without proper authorization.

The first queries check that authorization is possible under the model. We expect to see traces of the sort: "The file is readable by the attacker if the user has sent a read URI permission to the attacker" or "A file in private storage is readable by the attacker if the application makes the file world readable, the path world executable, and sends an intent to the attacker with the path to file." Those represent valid authorization paths in the model.

The second queries ensure that there are no other ways to read or write files aside from the authorization paths defined. If ProVerif finds any such paths, they are attacks. For example, ProVerif points out that line 39 above leaks the file to the attacker since it is allowed to read files on the SD card.

5 Authzoid Implementation

As shown above, the proper use of the authorization tools in Android requires careful design and analysis. In this section we describe our implementation of Authzoid, an app that lets file owners define authorization policies and then enforces them on their behalf. Authzoid uses the authorization tools explained in Section 2 to enable a wide variety of policies, including ones far richer than the minimal policy defined in Section 3.1. Authzoid is useful as a sample implementation of proper authorization, fixing the mistakes of the apps discussed above

and can be useful as a starting point for developers who want to get authorization right. Its source code can be found at: `http://prosecco.gforge.inria.fr/Essos/pv/`.

Authzoid offers three application-facing interfaces: file submission, policy definition, and file retrieval. It manages file versions and authorization checks internally.

Submission Interface Applications can submit files to Authzoid for storage using an intent with a custom action. The intent can contain a file to share or a content URI to resolve. Files can be submitted as new or as updates to existing files.

If submitted as new, Authzoid retrieves the name of the submitting application via the Android API and stores it in its private storage area. A private database indexes the files by their original file name or URI and submitting application. The new file is assigned a new version number which is returned to the submitter. The submitter may optionally include a permission as a string extra. If included, any application with the given permission may later read or modify the file (see below).

If submitted as an update, the file must be accompanied by the name of the owner, the file's original path and file name, and a version number which indicates the last version of the file the submitter saw. Authzoid first checks if the submitter is authorized to update the file (see below). If not, an authorization failure message is returned. If the update is authorized, but the version number submitted is smaller than the current version number in the database, the update is rejected with an explanatory message. Otherwise, the file is copied in to private storage and the database is updated. Authzoid generates a new version number which it stores in the database and sends it back to the submitter.

Policy Definition By default, only the application which submitted a file (its owner) can read or write it. Authzoid enables owners to share the file via URI permissions, by adding another app to the file's read and write access control list, or by retrieving a read-only randomized path name (similar to Google Drive). Groups of apps can be added by setting a permission on the file; then any application with the permission can read or write the file.

Retrieval Interface An app can request a file from Authzoid by sending an intent with owner's name, the file's original path, and name. For each request, Authzoid queries its access control matrix to see if the requestor is authorized to read the file. If the read is approved, Authzoid checks if a copy of the latest version of the file is already in the cache. If not, it generates a new directory under its private `filecache` directory with a 128-bit random name and puts a copy of the file in it. The file and random directory are set to be world readable and the directory is set to be world executable. Whether new or existing, the full file path of the file are returned to the requestor using an intent with the full path in an extra.

When an application resolves a URI using Authzoid's content provider, the content provider makes a new copy of the file, opens a new file descriptor on it, deletes the file using the Java file API, and then returns the file descriptor. This

prevents read/read conflicts on the file. Since the file descriptor acts as a hard link, the Android OS will preserve the contents of the file until the recipient closes the file descriptor or is killed.

Folder listings can be requested by sending the owner's name and the path via an intent. Authzoid checks its access control matrix to see if the requestor is the owner or authorized to list the directory. If authorized, a listing of all files and directories in and under the given directory is sent back via an intent as a string array extra.

Authzoid is the first Android app that provides a unified authorization service enabling file sharing between Android apps. Using ProVerif, we verified that Authzoid is secure against the class of attacks captured in our formal model. This should not be interpreted as a formal theorem however, since our model of both Android is abstract and incomplete, and may hide other attacks. Still, our analysis presents a first step towards formal security analysis for Android applications. Our models are public and may be extended for more sophisticated analysis.

6 Related Work

Research on Android's security infrastructure includes studies on how permissions are enforced [17], used [2], and misused or attacked [10, 12, 18, 22]. Some try to secure Android applications against attackers by performing static or dynamic analysis of apps (*ex.* [16, 8, 20]). Xu, *et al.* [29] developed Aurasium, a tool that uses static analysis and code injection to detect or prevent privilege escalation attacks. Like Authzoid, Aurasium does not require modifications to the operating system. Conti, *et al.* [11] developed CRePE, a system capable of enforcing rule based context aware security policies. Naumann, *et al.* [23] extended Android permission with custom user defined constraints. None of the above work includes formal analysis or verification.

Research on formalization of the Android stack and API includes Chaudhuri [9] who gave a formal model of a subset of the Android communication system; Enck, *et al.* [15] who developed TaintDroid to track the flow of sensitive information between Android apps (extended by Shreckling, *et al.* [25] with more complicated, dynamic run time policies); and Armando, *et al.* [1] who presented a more complete model of the Android middleware using types.

With respect to formalizations for secure sharing of resources, Blanchet and Chaudhuri [6] developed a formally verified protocol for secure file sharing on untrusted storage (a tool which could be used to secure Android's SD card) and Fragkaki, *et al.* [19] gave formal typing rules to explain Android's security model. Similar to our work, Fragkaki *et al.* described Sorbet, a modification to Android which enforces secrecy and integrity properties written by app developers. In contrast, Authzoid is developed to enable the easy specification of authorization policies and relies upon existing Android mechanisms without requiring changes to the operating system.

Since many Android apps are distributed free and make money from in-app ads, work has been done to determine how ad libraries operate and whether they pose privacy or authorization risks. Dietz, *et al.* [13] developed Quire which enabled advertisers to prevent app based ad fraud. Stevens, *et al.* [27] investigated the behavior of thirteen ad libraries and showed how their requirements cause app developers to request more permissions than necessary (*permission bloat*). As a remedy to permission bloat, Shekhar, *et al.* [26] implemented AdSplit, a mechanism to separate ad libraries from individual apps.

7 Conclusion

Many Android apps attempt to enforce authorization policies for sharing resources, but fail due to misuse of the Android authorization tools or due to actions by external entities. We can discover authorization attacks by using ProVerif to model a relevant subset of the Android authorization tools and environment and use it to examine the behavior of sharing applications. We also describe Authzoid, an application which lets app developers specify authorization policies for sharing and enforces them using built-in Android tools. Future extensions to Authzoid include work on making an encrypted cache on the SD card and enabling it to proxy OAuth based web sharing.

References

1. A Armando, G Costa, and A Merlo. Formal modeling and reasoning about the android security framework. In *7th Intl Sym on Trustworthy Global Computing*, 2012.
2. D Barrera, H G Kayacik, P C van Oorschot, and A Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *17th ACM Conf on Computer and Comm Security (CCS '10)*.
3. A. Belenko and D. Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Ltd., 2012.
4. K Bhargavan and A Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *6th USENIX Workshop on Offensive Technologies (WOOT'12)*.
5. B Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop (CSFW'01)*.
6. B Blanchet and A Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Sym on Security and Privacy (SP '08)*.
7. Tim Bray. Recent Android app update prevents third-party apps from using com.google.android.gm.permission.READ_GMAIL. Why? `productforums.google.com/d/msg/gmail/XD0C4sw9K7U/8KwuZl0Rl68J`, 29 July 2011.
8. P P F Chan, L C K Hui, and S M Yiu. Droidchecker: analyzing android applications for capability leak. In *ACM Conf on Security and Privacy in Wireless and Mobile Networks (WISEC '12)*.
9. A Chaudhuri. Language-based security on android. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*.
10. PH Chia, Y Yamamoto, and N Asokan. Is this app safe? A large scale study on application permissions and risk signals. In *WWW '12*.

11. M Conti, V Nguyen, and B Crispo. Crepe: context-related policy enforcement for android. In *13th Intl Conf on Information Security (ISC '10)*.
12. L Davi, A Dmitrienko, A Sadeghi, and M Winandy. Privilege escalation attacks on android. In *13th Intl Conf on Information Security (ISC '10)*.
13. M Dietz, S Shekhar, Y Pisetsky, A Shu, and D Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Conf on Security*, 2011.
14. E. Hammer-Levy (ed.). *The OAuth 2.0 Authorization Protocol*. IETF, 22 Sept 2011. draft-ietf-oauth-v2-22. Work in Progress. Expires 25 March 2012.
15. W Enck, P Gilbert, B Chun, L Cox, J Jung, P McDaniel, and A Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Conf on Operating Systems Design and Implementation (OSDI '10)*.
16. W Enck, M Ongtang, and P McDaniel. On lightweight mobile phone application certification. In *16th ACM Conf on Computer and Comm Security (CCS '09)*.
17. A Felt, E Chin, S Hanna, D Song, and D Wagner. Android permissions demystified. In *18th ACM Conf on Computer and Comm Security (CCS '11)*.
18. A Felt, H Wang, A Moshchuk, S Hanna, and E Chin. Permission re-delegation: attacks and defenses. In *20th USENIX Conf on Security (SEC'11)*.
19. E Fragkaki, L Bauer, L Jia, and D Swasey. Modeling and enhancing android's permission system. In *ESORICS 2012*.
20. A Fuchs, A Chaudhuri, and JS Foster. SCanDroid: Automated security certification of android applications. Technical report, U of Maryland College Park, 2009.
21. Google. *Android 4.1 Compatibility Definition*. Android Compatibility Program, 7 Sep 2012. Rev 2.
22. P Hornyack, S Han, J Jung, S Schechter, and D Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM Conf on Computer and Comm Security (CCS '11)*.
23. M Nauman, S Khan, and X Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symp on Information, Computer and Communications Security (ASIACCS '10)*.
24. NielsenWire. State of the appnation a year of change and growth in u.s. smartphones. `blog.nielsen.com/nielsenwire/online_mobile/state-of-the-appnation-\%E2\%80\%93-a-year-of-change-and-growth-in-u-s-smartphones/`, 16 May 2012.
25. D Schreckling, J Posegga, J Köstler, and M Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *6th IFIP WG 11.2 Intl Conf on Inf Security Theory and Practice (WISTP '12)*.
26. S Shekhar, M Dietz, and D Wallach. Adsplit: separating smartphone advertising from applications. In *21st USENIX Conf on Security (SEC '12)*.
27. R Stevens, C Gibler, J Crussell, J Erickson, and H Chen. Investigating user privacy in android ad libraries. In *MoST 2012: Mobile Security Technologies*.
28. K Varma. Security permissions in android. Accessed 9 Oct 2012, Krishnaraj Varma's Blog, 3 Oct 2010. `www.krvarma.com/2010/10/security-permissions-in-android/`.
29. R Xu, H Saïdi, and R Anderson. Aurasium: practical policy enforcement for android applications. In *21st USENIX Conf on Security (SEC '12)*.